ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Essentials of Embedded Software Engineering

Christian Plessl

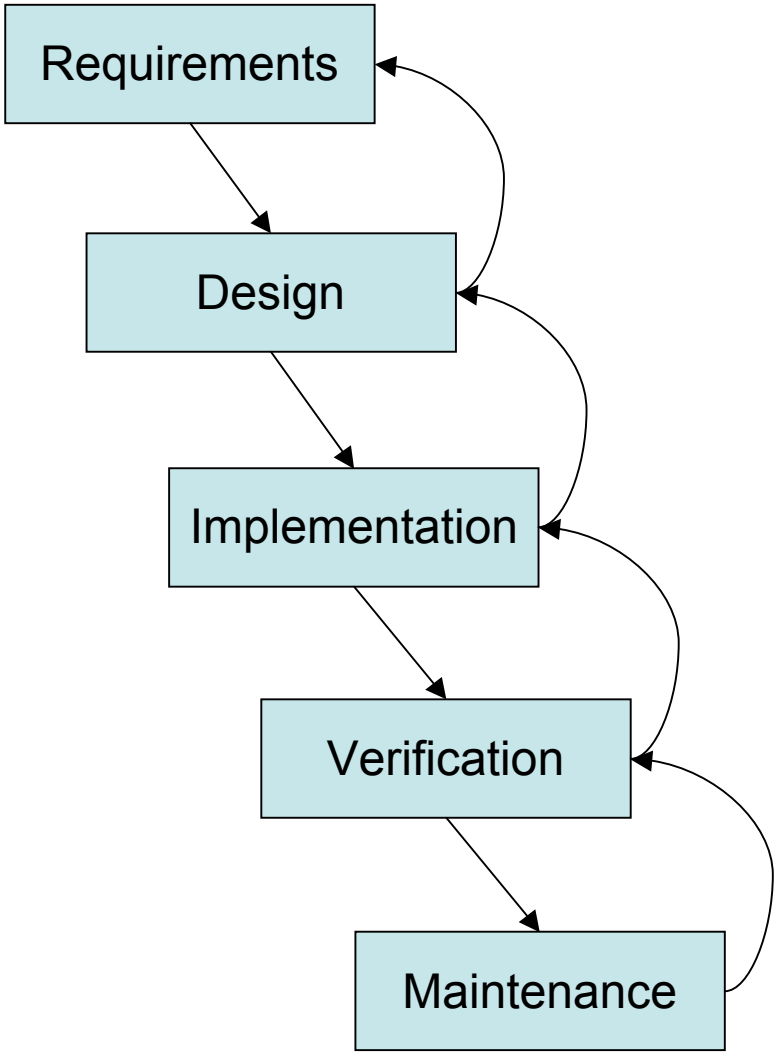
MICCS Developer Workshop

November 20, 2006

outline

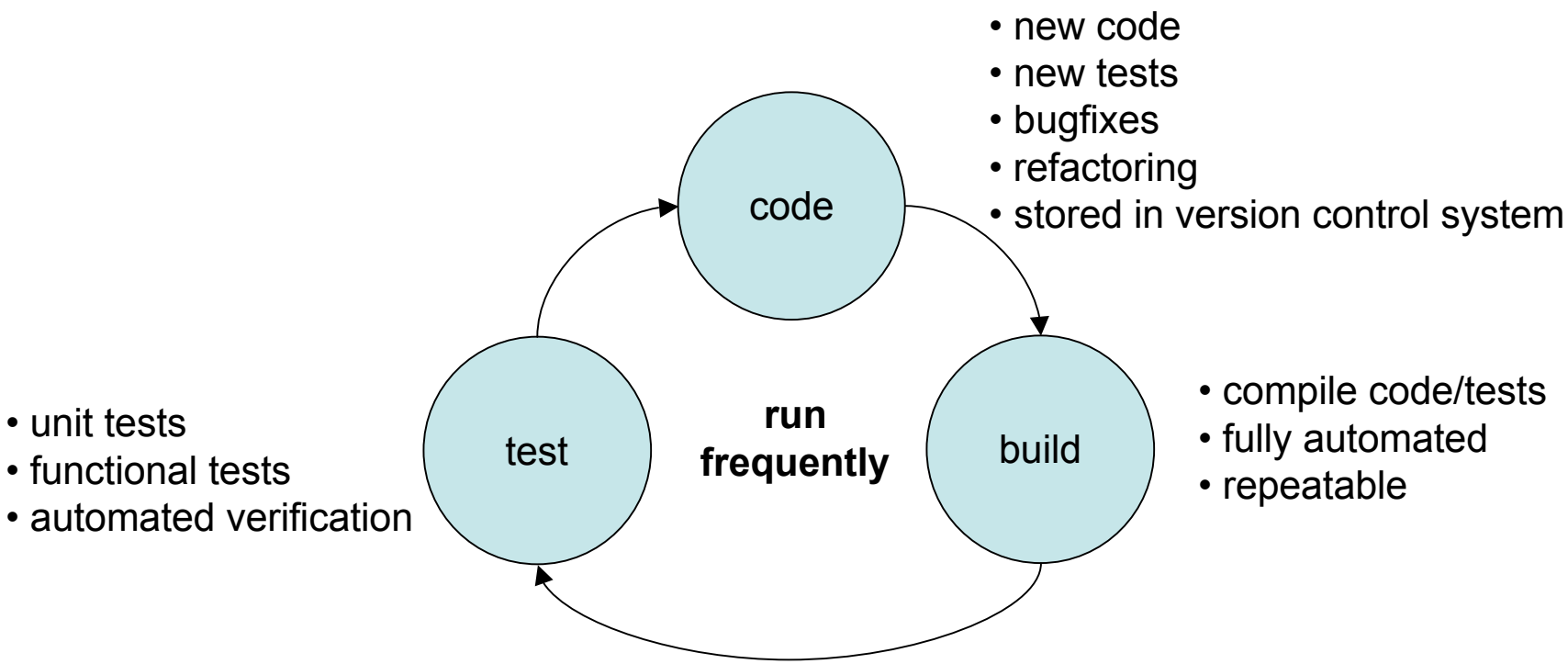
- introduction
- testing
- building
- version control

waterfall model



- flaws
  - requirements change
  - results only very late
  - implementation difficulties can arise late in the process
  - long feedback cycles
  - software development is art not just engineering
- the pure waterfall model is dead
- replaced by other paradigms
  - Extreme Programming
  - Agile development
  - UML design
  - ...

agile development cycle



agile development for academic software

- academic software development
  - rather small projects
  - requirements constantly changing
  - limited experience of programmers
  - integration of work from collaborators (students, project partners)
  - different versions of software (customized for demonstrations, publications)
- agile development matches our requirements well
  - focus on working software, not on processes
  - monitor the software development process
  - promote change by reducing risk for introducing bugs

best practices applicable to embedded sw

- version control
  - support a global “undo”
  - document changes and reason for changes
- automated building and testing
  - repeatable, remove human from the loop
  - speed-up build-test-cycle to promote frequent testing
  - prevent regression, improve confidence in code
- iterative software refinement
  - adapt to changing requirements
  - clean-up code (refactoring)

testing

- tests ...
  - ... monitor the software quality
  - ... prevent regression bugs
  - ... explicitly state assumptions and expectations
  - ... serve as examples and documentation
- test early, test often, test automatically
  - tests should state expected behavior
  - compare expected with observed behavior (built-in self test)
- levels
  - unit testing
  - integration testing
  - performance testing

unit testing

- test functionality of code units in isolation
- implementations
  - many existing frameworks
  - extensive list: http://www.opensourcetesting.org/
  - build your own framework based on assertions
- what to test: the Right BICEP rule
  - are the results **right**?
  - are all boundary conditions **correct**?
  - can you check **inverse** relationships?
  - can you **cross-check** results using other means?
  - can you force **error conditions** to happen?
  - are **performance** characteristics within bounds?

unit testing - example JUnit

```
public class Largest {
 public static int largest(int[] list) {
  int index, max=0;
  for (index = 0; index < list.length-1; index++) {
    if (list[index] > max) { max = list[index]; }
  }
  return max;
 }
}
```

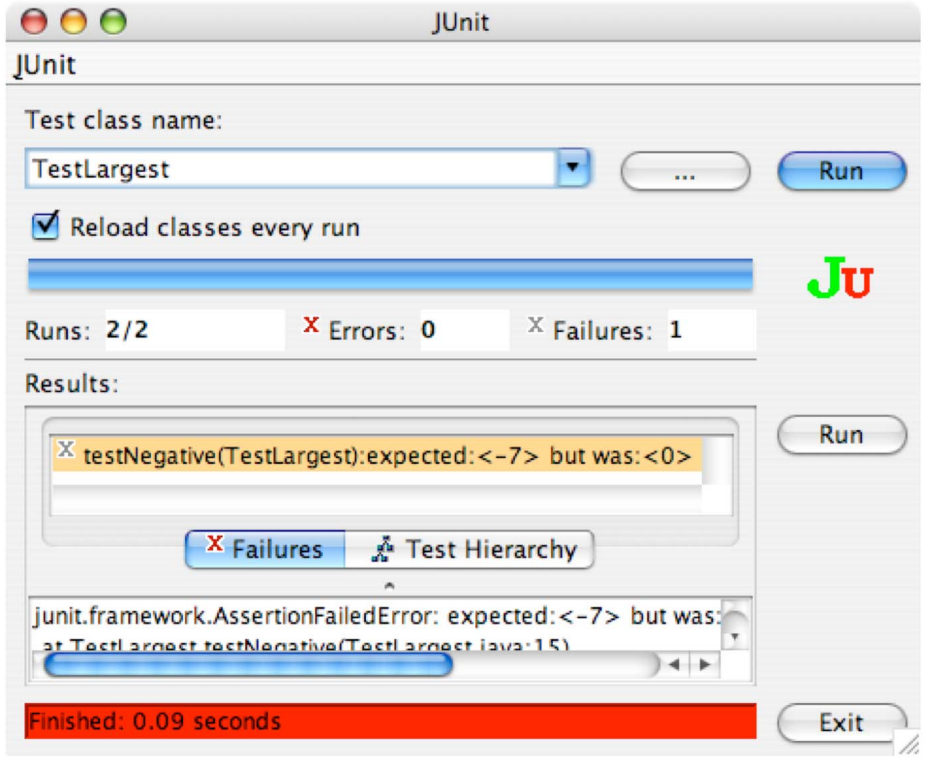
class to be tested

```
import junit.framework.*;

public class TestLargest extends TestCase {
  public TestLargest(String name) {
    super(name);
  }
  // initialize/destroy data structures for tests here
  public void setup(){ /* ... */ }
  public void tearDown(){ /* ... */}

  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {8,9,7}));
  }
  public void testNegative() {
    assertEquals(-7, Largest.largest(new int[] {-9, -8, -7}));
  }
}
```

unit test

unit testing - example JUnit (2)

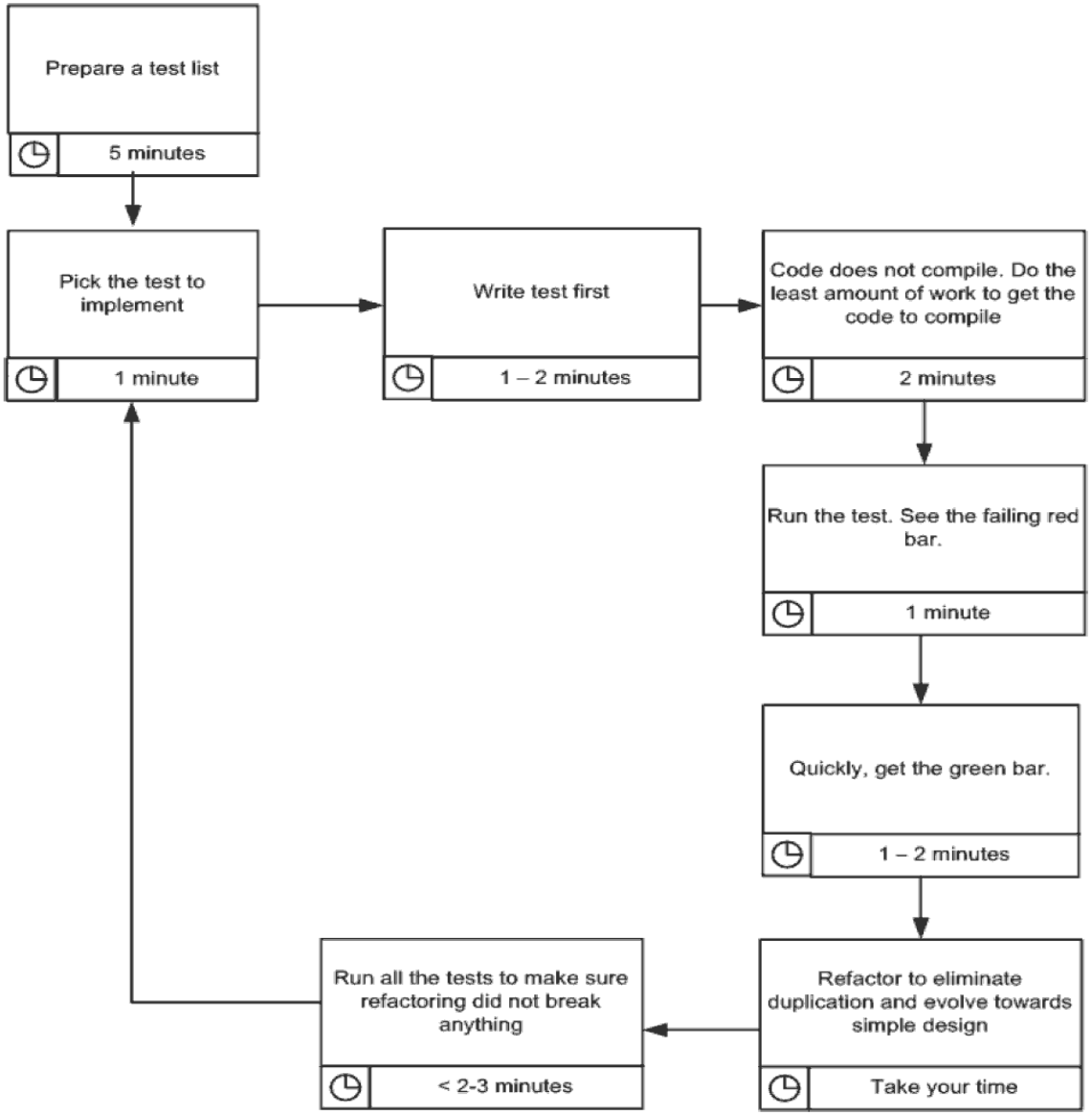
```
[Christian-Plessls-Computer:unittesting]$ make test
javac -classpath 'junit.jar:.' Largest.java
javac -classpath 'junit.jar:.' TestLargest.java
java -classpath 'junit.jar:.' junit.textui.TestRunner TestLargest
..F
Time: 0.016
There was 1 failure:
1) testNegative(TestLargest)junit.framework.AssertionFailedError:
   expected:<-7> but was:<0>
[....]
FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```



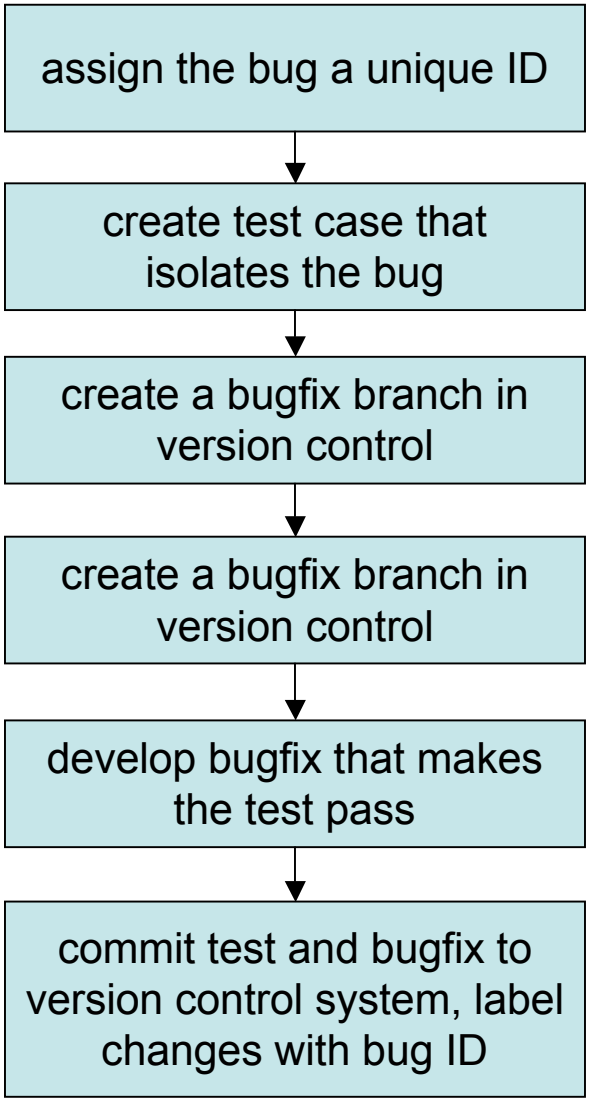
functional / integration testing

- combine individual software components
  - bottom-up approach
  - assemble unit-tested components
  - show the components work well together
  - validate functional properties of whole software

test-driven development

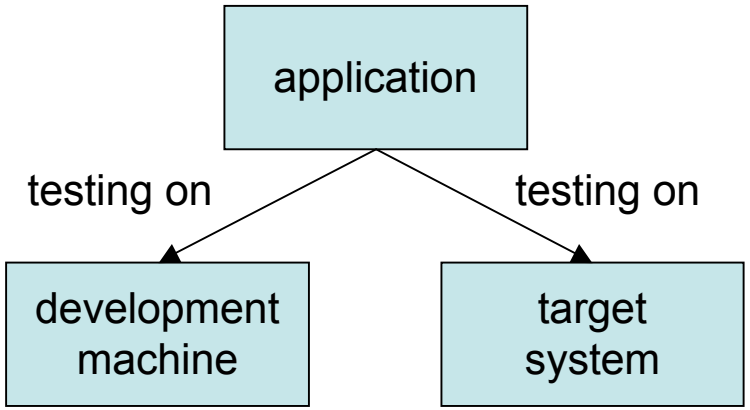


when a (non-trivial) bug is found



- keep track of software defects
- deal with each bug only once
- prevent regression

testing without target hardware

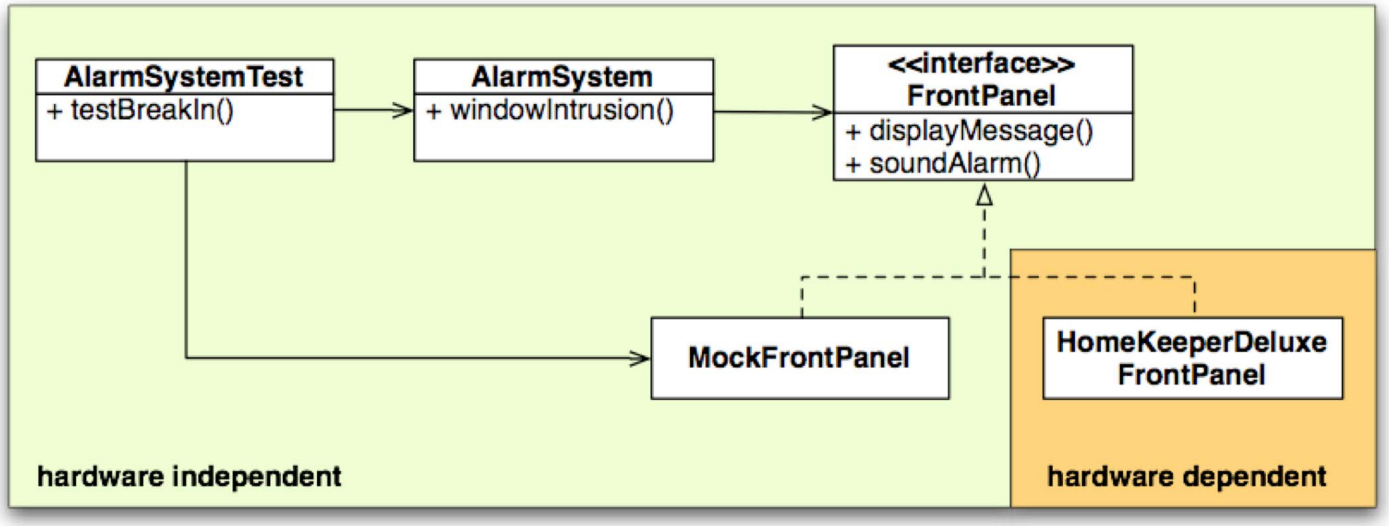


- fast
- lots of memory
- good debuggers

- slow execution
- limited resources
- difficult to debug
- slow reprogramming
- **special hardware (IO, radio,...)**

- testing on developer machine is convenient
  - fast compile/test cycles
  - good debuggers
- how to test target software on development machine despite of missing special hardware?
  - test only hardware independent parts
  - model hardware with mock objects
  - restrict models to minimum

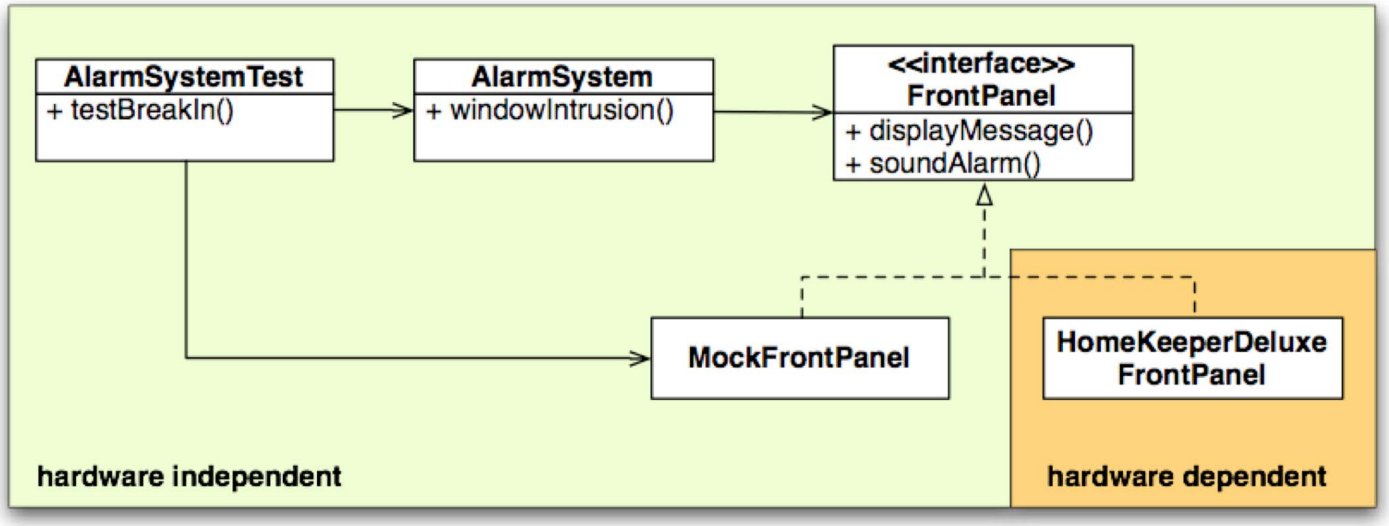
mocking special hardware




```
TEST (HomeGuard, WindowIntrusion)
{
 MockAlarmPanel* panel = new MockAlarmPanel();
 Homeguard hg(panel);
 hg.arm();
 hg.windowIntrusion();
 CHECK(true == panel->isArmed());
 CHECK(true == panel->isAudibleAlarmOn());
 CHECK(true == panel->isVisualAlarmOn());
 CHECK(panel->getDisplayString() == “Window Intrusion”);
}
```

example from: James Grenning, “Progress before Hardware”, Object Mentor

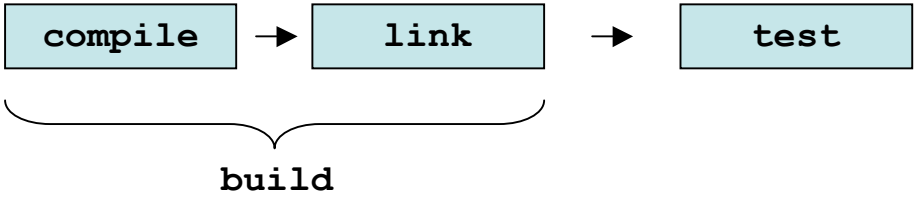
mocking special hardware




```
class MockFrontPanel : FrontPanel
{
 private _audibleAlarm;
 public MockFrontPanel(){...}

 displayMessage(){...}
 soundAlarm(){_audibleAlarm = 1;}
 bool isAudibleAlarmOnsetAudibleAlarmOn(){
  return _audibleAlarm
 }
 ...
};
```

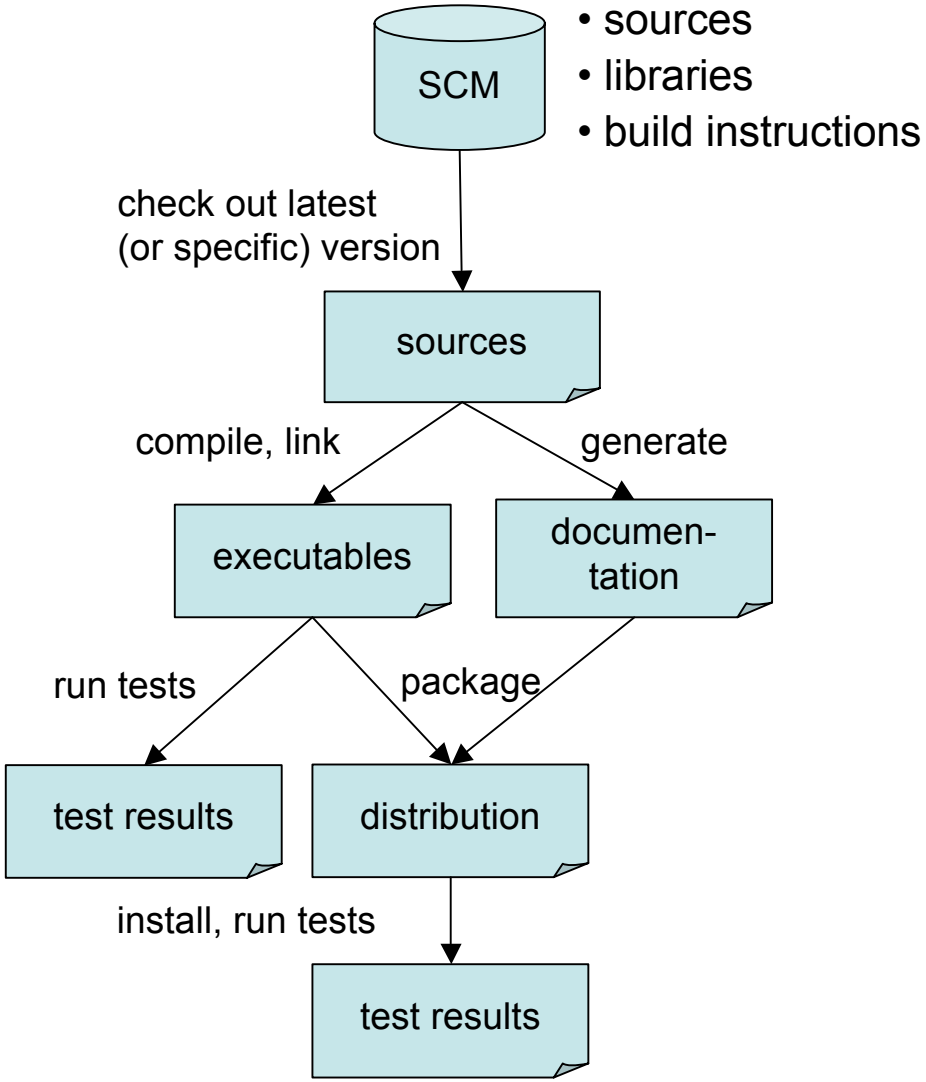
building and testing software

- basic steps



- strategy
  - only do what is necessary (modification time)
  - store build recipe in one place
  - use build tools (make, ant, ...)
- benefits
  - repeatable builds
  - incremental builds (faster)
  - can be triggered by changes => continuous integration
- why not using the IDE's build process
  - can do the same with one click
  - information distributed, not portable
  - some IDEs can export build procedure as makefile

an exemplary build process



- building and packaging fully automated
- all sources, external libraries, and build instructions are under version control
- ability to build
  - the current software version
  - any previous software version
- same build system for
  - development, testing
  - releasing

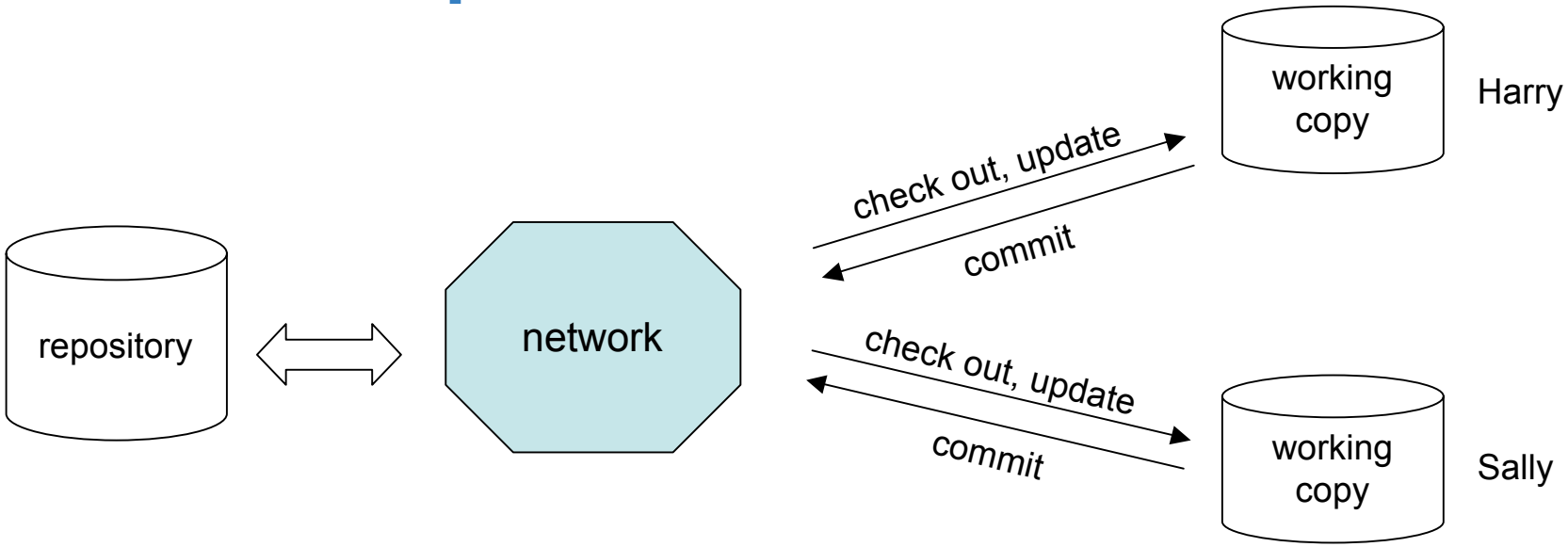
build languages

- make, ant, SCons, rake
- domain knowledge
  - hierarchy of build targets
  - build rules (templates e.g. file.java → file.class)
  - dependencies
  - minimal incremental rebuilds
- general purpose or language specific
- example ant
  - knows about javac, jar, rmic, javah, javadoc, etc.
  - tasks: archive, coverage, compile, deployment, execution, logging, scm, test, etc

why version control?

- version control = time machine + multi-user access
- history
  - store snapshots of all files
  - track and document changes between snapshots
  - access to all previous versions
- multi-user management
  - enable concurrent modifications of shared files
  - detect and resolve conflicts
  - enable remote access
- tools
  - CVS, Subversion, many more
  - recommendation: Subversion
  - command-line and graphical clients

basic concepts

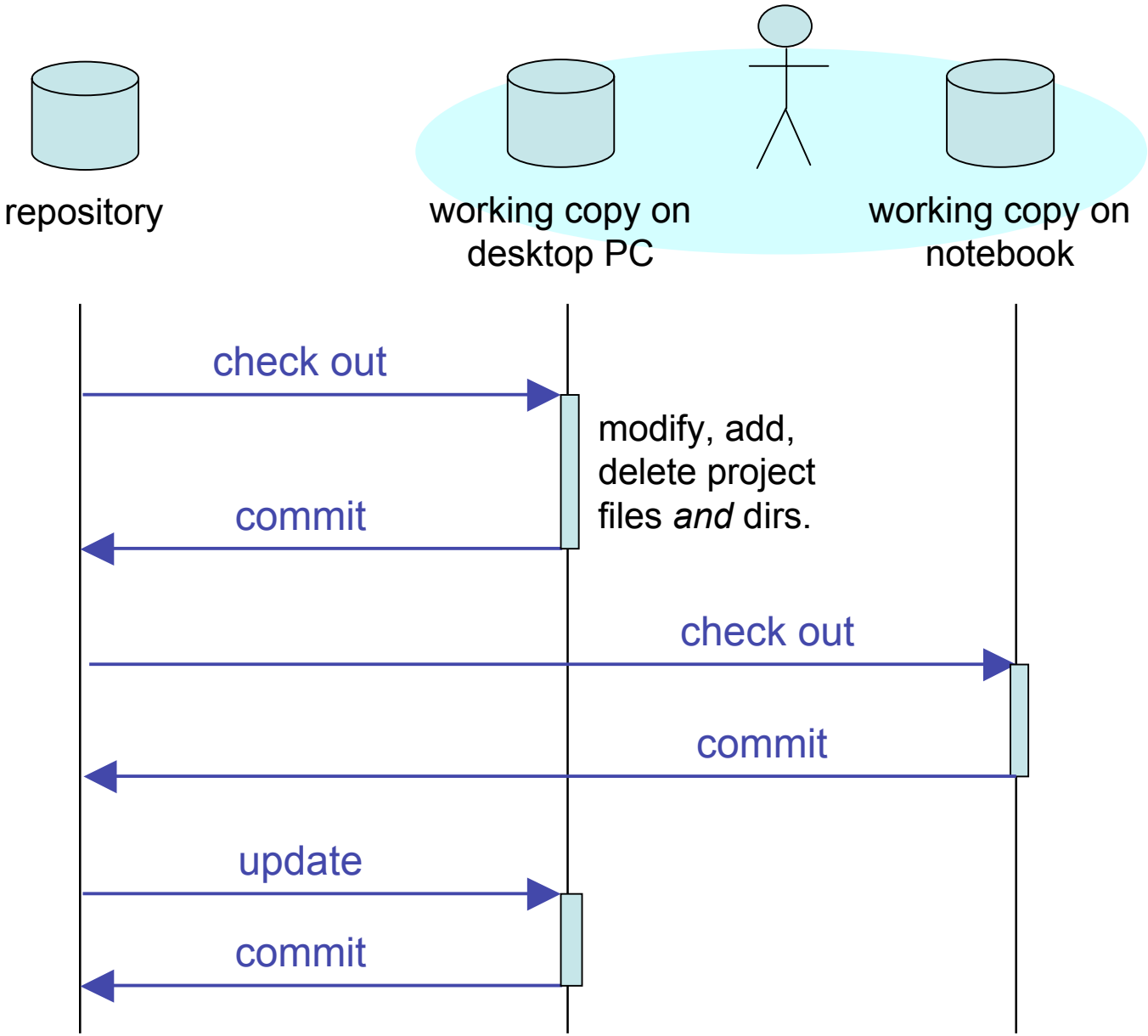


- **repository** is a central place that holds the master copy of all versions of project's files.
- **working copy** is a local copy of all of the things that a user needs from the repository to work on its part of the project.

example 1: single user project

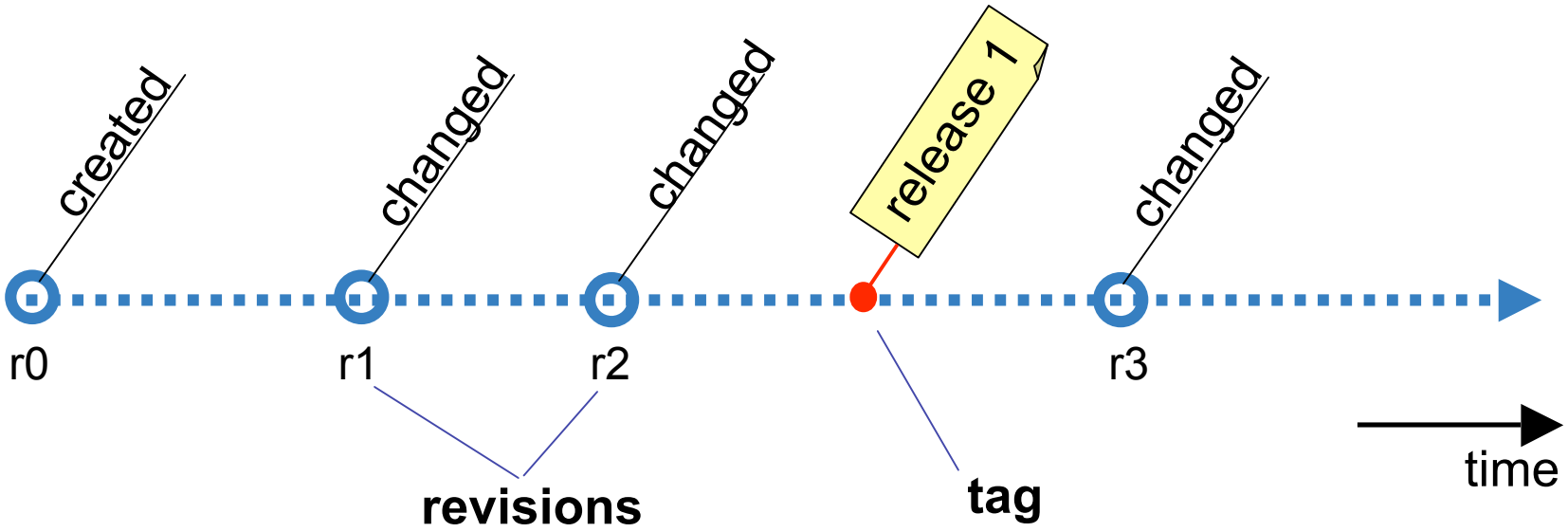
- problem
  - how to track changes made to the project files?
  - how to roll back in case of a “false move”?
  - how to synchronize project data on different computers (e.g. between a laptop and a desktop)?
- solution with version control
  - revisions, tags
  - networking

scenario


repository
working copy on desktop PC
working copy on notebook
check out
modify, add, delete project files *and* dirs.
commit
check out
commit
update
commit

tracking changes

- **revision** is a group of committed changes. Each revision is assigned a unique number.
- **tag** is a name assigned to a group of files at a particular point in time. It indicates a significant point in the development.



creating a repository

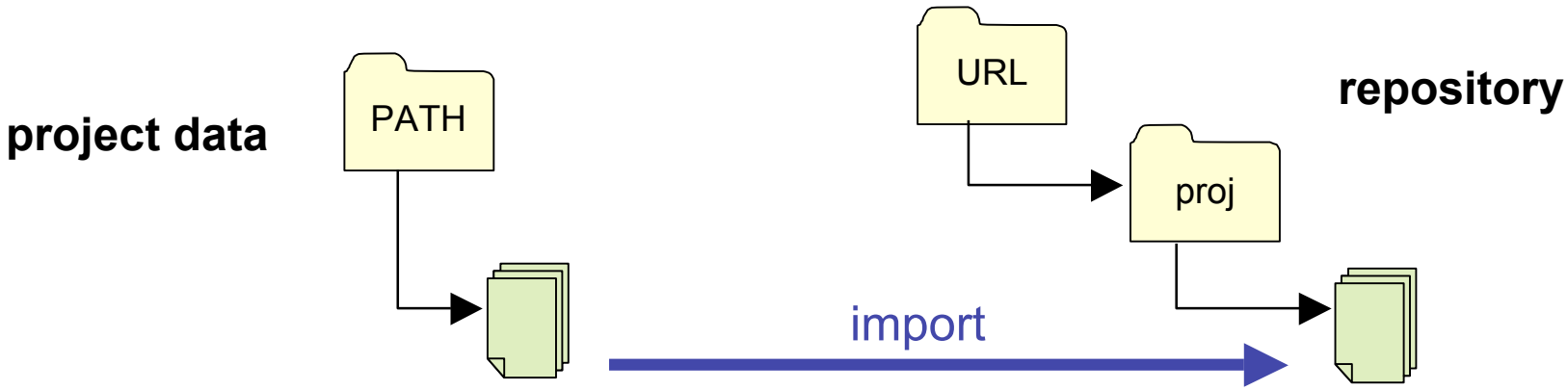
```
svnadmin create <PATH>
```

creates a new repository in the directory PATH.

importing project files into a repository

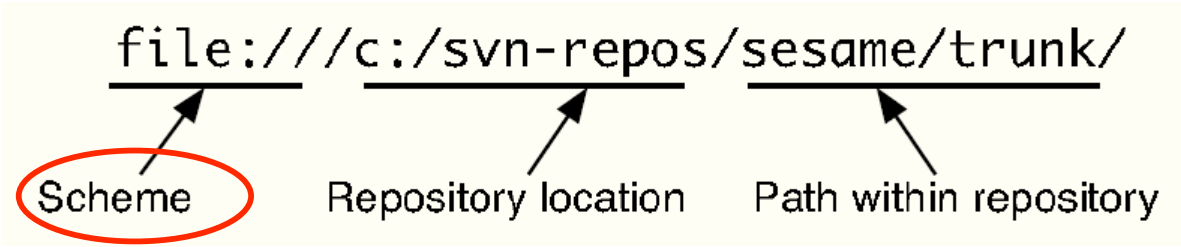
```
svn import <PATH> <URL/proj>
```

copies project files *and* directories located at `PATH` into the repository at `URL/proj`.



accessing a Subversion repository

- repository identified with URL



- supported schemes
  - **file:** access a repository on the local file system
  - **svn:** SVN protocol, most suitable for a team on the same LAN
  - **svn+ssh:** SVN tunneled through SSH for strong encryption
  - **http:** access a repository hosted by the Apache web server; a wide range of authentication options is available

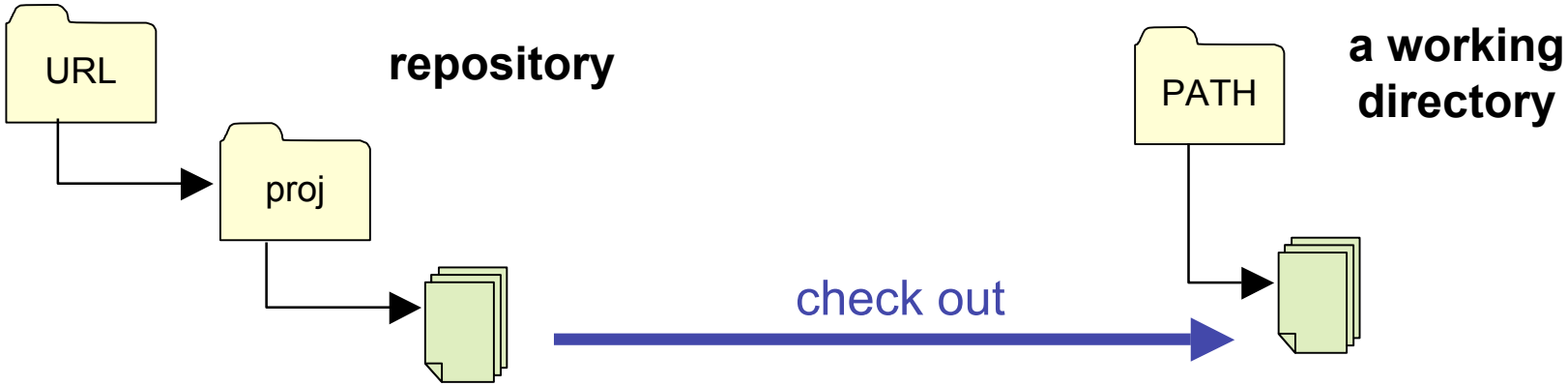
checking out a revision

```
svn checkout <URL/proj> <PATH>
```

checks out the latest revision from the repository at `URL/proj` into the directory `PATH`

```
svn checkout -r N <URL/proj> <PATH>
```

checks out revision `N`



updating a working copy

```
svn update <PATH>
```

brings changes from the repository into your working copy (located at `PATH`)

committing changes

```
svn commit -m "a log message here" <PATH>
```

sends changes from your working copy (located at `PATH`) to the repository

- changes are local to working copy
- permanently stores changes in repository
- commit small changes
- log messages
  - details about a set of changes
  - also important for communication within a team

changed timeout to 42	If the round-robin DNS returns a machine that is unavailable, the connec() method attempts to retry for 30ms. In these circumstances our timeout was too low
a bad log message	a good log message

examining the change history

`svn log`

displays the log messages for all the paths that changed in your working copy

`svn log <FILE>`

displays the log messages for `FILE` in your working copy

`svn log -r <RA>:<RB>`

displays the log messages for revisions in the range `[RA,RB]`

example:

```
sesame> svn log -r 19:24 Clock.java
------------------------------------------------------------------------
r19 | mike | 2004-10-04 21:47:09 -0600 (Mon, 04 Oct 2004)
Renamed util to common
------------------------------------------------------------------------
r21 | mike | 2004-10-09 16:33:00 -0600 (Sat, 09 Oct 2004)
Fixed compilation problems
------------------------------------------------------------------------
r22 | dave | 2004-10-09 16:48:23 -0600 (Sat, 09 Oct 2004)
Added setTime() method
------------------------------------------------------------------------
r23 | ian | 2004-10-09 17:00:23 -0600 (Sat, 09 Oct 2004)
Added setTime() method taking a Date
------------------------------------------------------------------------
r24 | dave | 2004-10-10 18:07:08 -0600 (Sun, 10 Oct 2004)
Added Log class
------------------------------------------------------------------------
```

revision #

commit date

who made the change

log message

other useful Subversion commands

- **svn status** — Print the status of working copy files and dir's
- **svn diff** — Display the differences between versions
- **svn revert** — Revert any local changes to a file or directory
- **svn add** — Add files and directories
- **svn move** — Move a file or directory
- **svn delete** —Delete a file or directory
- …

example 2: small multi-user project

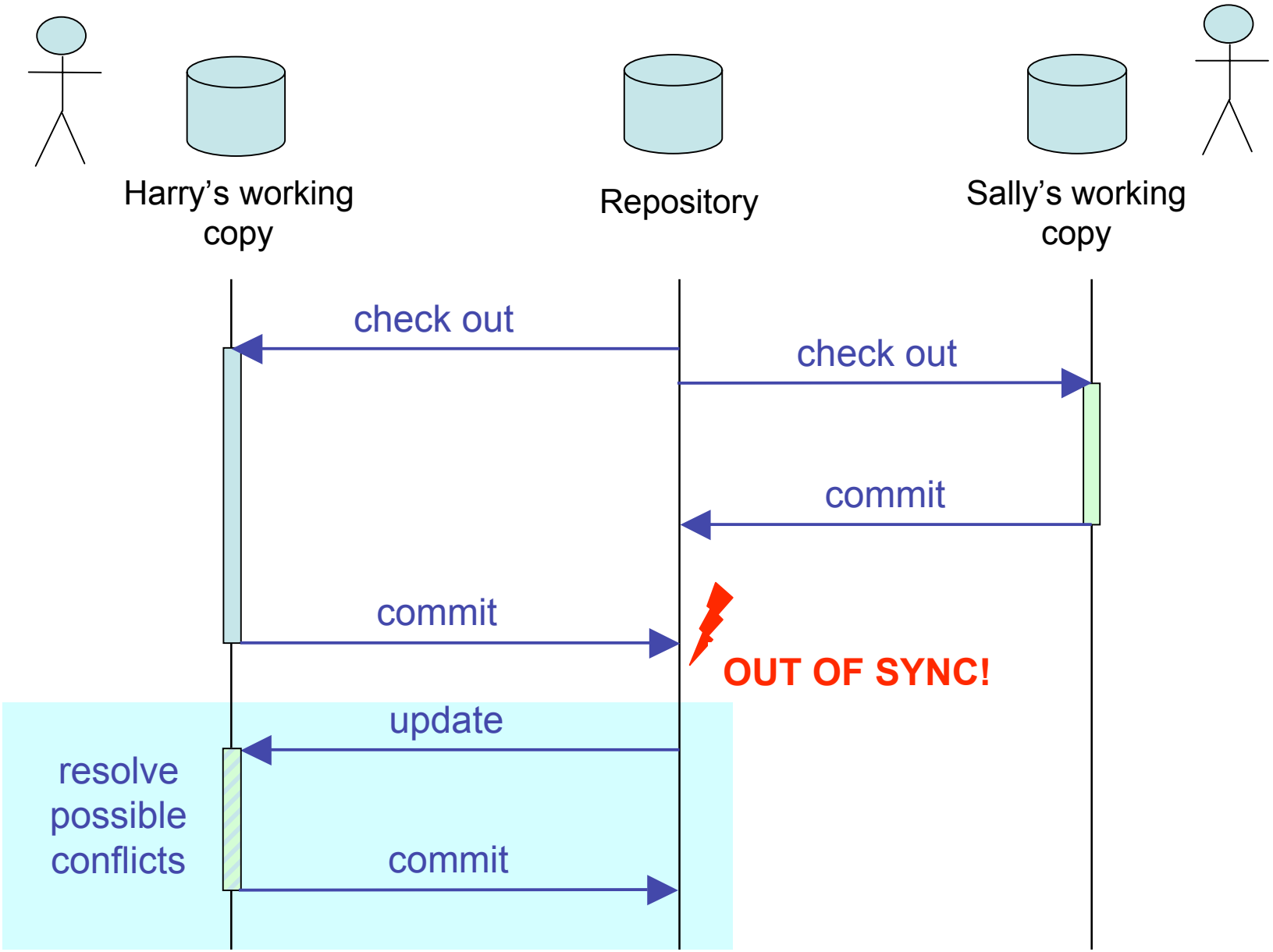
problem

- how to organize file exchange between the users?
- how to let them modify files (i.e. work) concurrently?
- how to merge contributions of all the users into a working whole?

solution with version control:

- update & merge
- support for resolving merge conflicts

copy-modify-merge



update & merge

- commit will fail if there is a need for update

Example:

```
$ svn commit --message "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: Out of date: 'rules.txt' in transaction 'g'
```

- update tries to merge the working copy with the repository version…

Example:

```
$ svn update
U   INSTALL
G   README
C   bar.c
Updated to revision 46
```

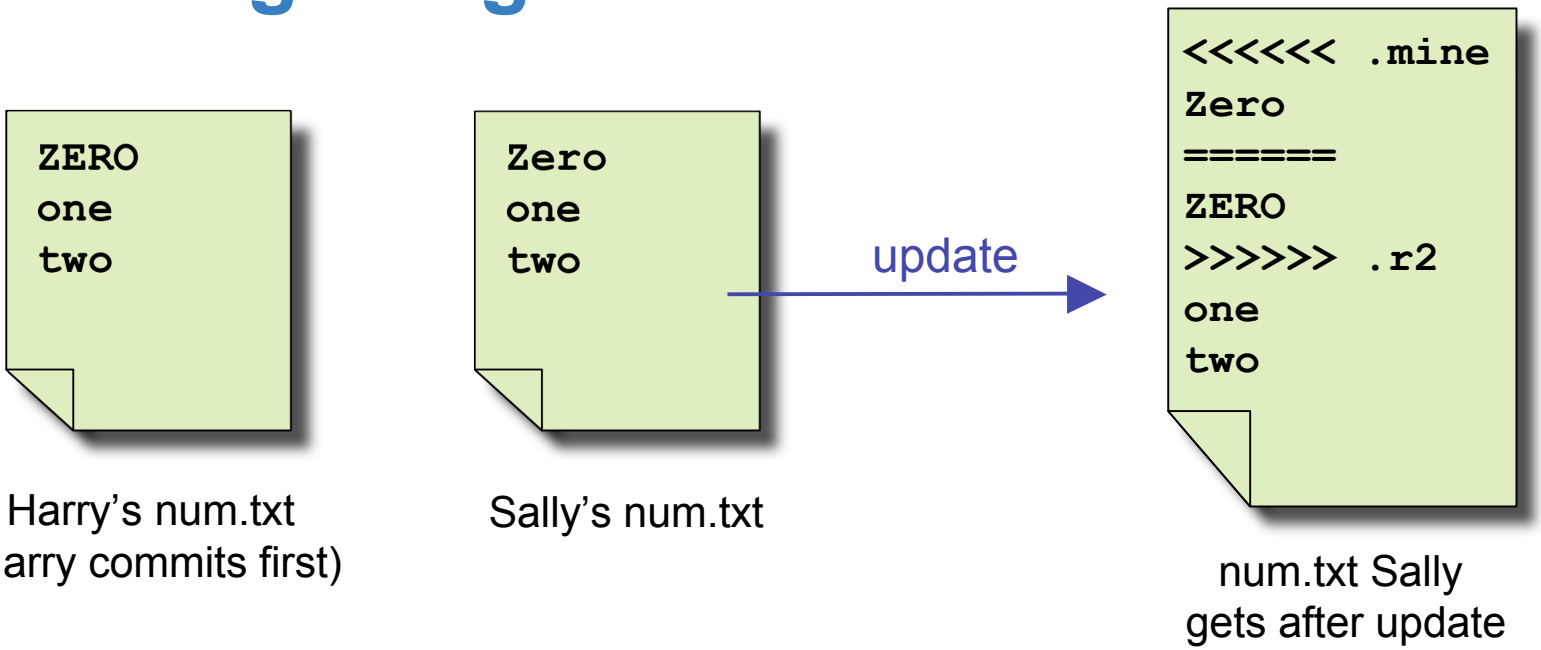
U – a successful update

G – a successful merge

C – conflict: requires resolution

…

handling merge conflicts



A

Sally rejects her changes:

```
$ svn revert num.txt
Reverted 'num.txt'

$ svn update num.txt
At revision 2.
```

B

Sally rejects Harry's changes:

```
$ cp num.txt.mine num.txt
$ svn resolved num.txt
Resolved conflicting
state of 'num.txt'
```

C

Sally does manual merge:

Edit num.txt

```
$ svn resolved num.txt
Resolved conflicting state
of 'num.txt'
```

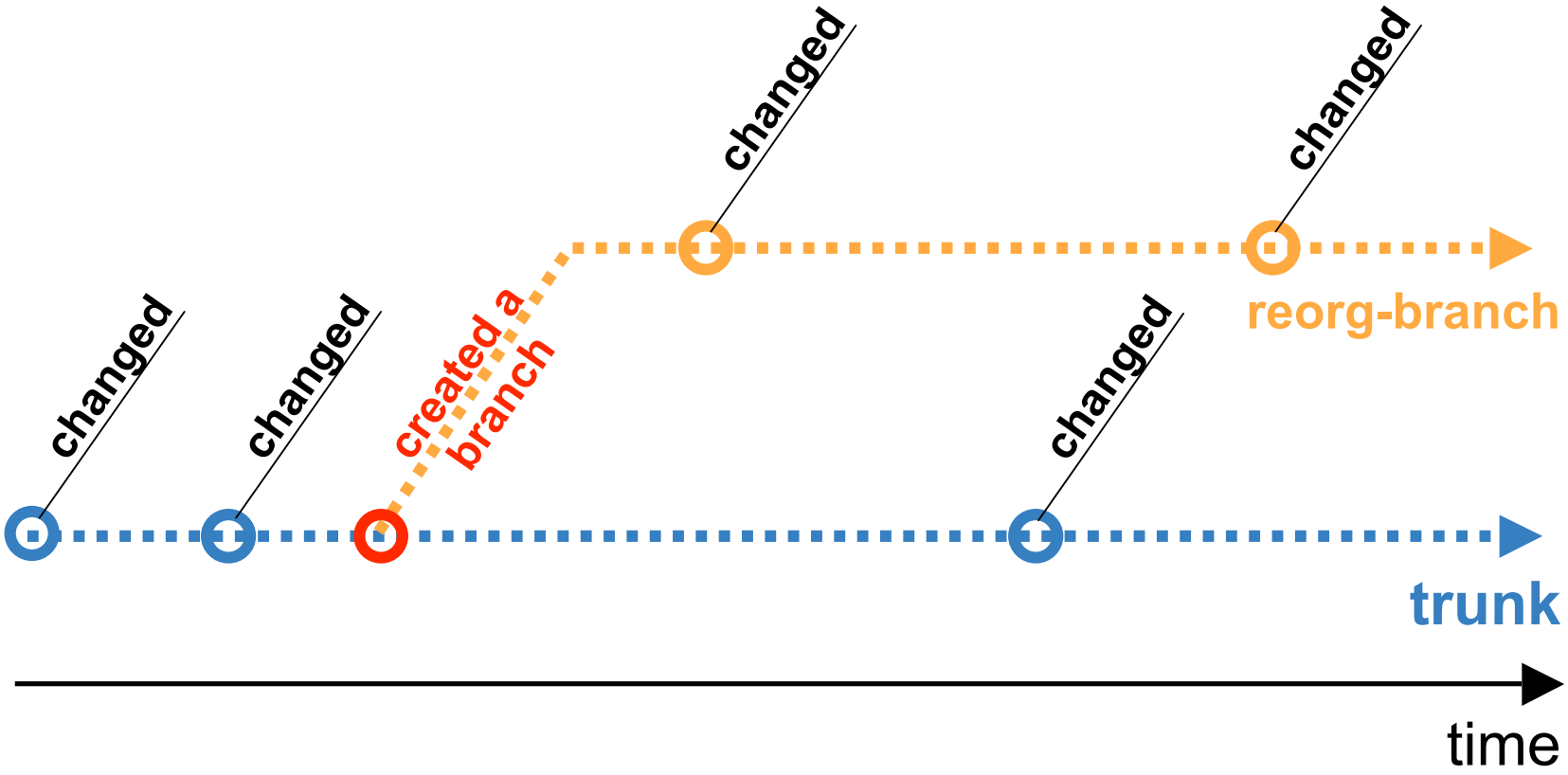
commit policies

- can help increase productivity
- define a clear commit policy in your team
  - what changes are committed to repository
  - who is allowed to commit
  - which branch is used to apply commits
- examples
  - commit only compilable versions
  - commit only tested versions (unit testing)
  - mandatory log messages with a predefined format
  - commit independent changes individually

example 3: multi-user long-term project

- problem statement
  - multiple users work concurrently on same project (e.g. software application/open source development)
  - software should be mostly stable at all times
  - perform long haul changes (e.g. bugfixes, redesign, refactoring) without interfering with other developers
- solution with version control
  - use branches
  - multiple concurrent lines of software development
  - isolate development of main branch from changes to other branches
  - can be merged into main branch after completion

branches


changed
changed
created a branch
changed
changed
changed
reorg-branch
trunk
time

creating a tag/branch with Subversion

tags and branches are "cheap copies" of files:

```
svn copy <url-source> <url-target>
```

suggested repository layout

Use the standard repository layout:

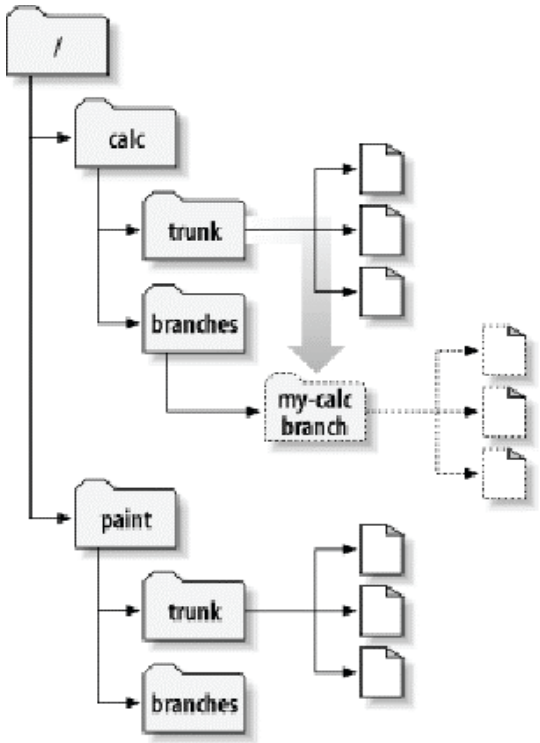
```
project1/
    trunk/          where you keep the files
    tags/           where you copy tags
    branches/       where you copy branches
```

Checkout trunk:

```
svn checkout <url>/project1/trunk
 project1
```

Make a branch:

```
svn copy <url>/project1/trunk \
        <url>/project1/branches/mybranch
```



merge command

- copy changes between branches
  - e.g. a small bug has been found and fixed in trunk: merge the change to the branches too
  - changes on a branch can be merged separately back to trunk
  - a whole branch can be merged back to trunk
- undo already committed changes

General svn merge syntax (one form):

```
svn merge -r <rev1>:<rev2> <source-url>
```

`source-url` is compared as it existed between `rev1` and `rev2`. Changes are applied to current directory (working copy).

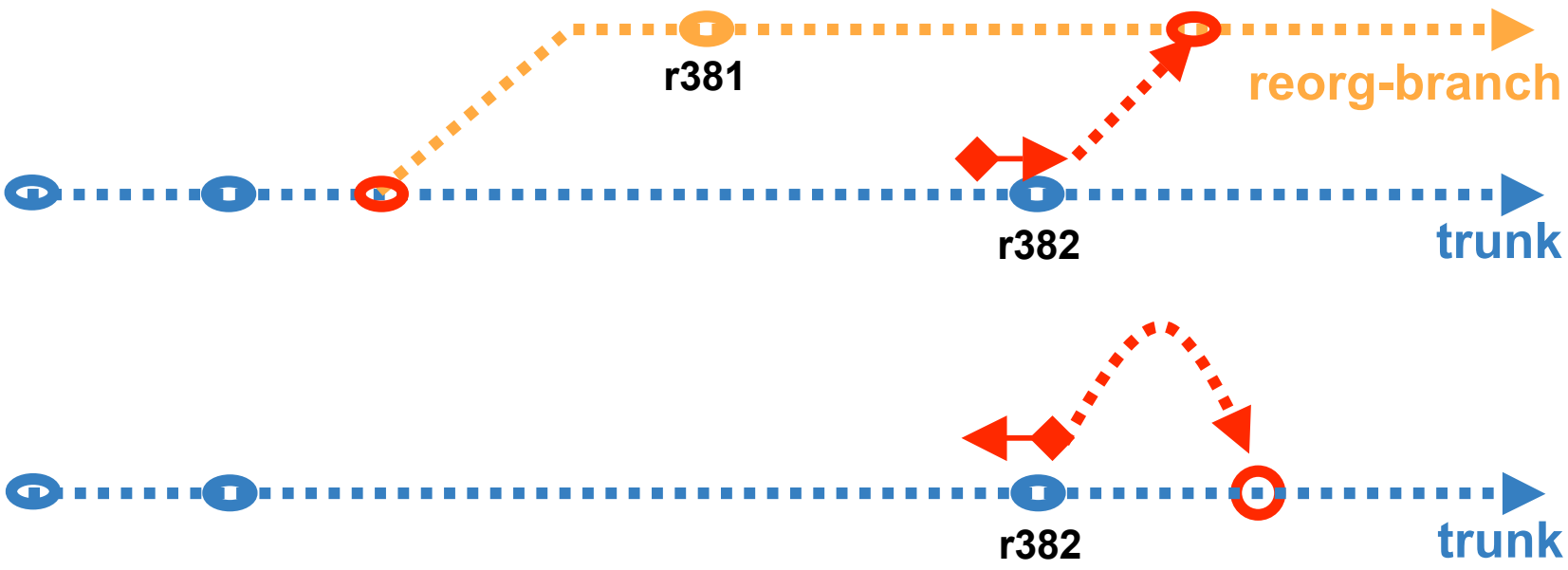
examples for merge command

porting a specific change in the trunk (r382) to a branch:

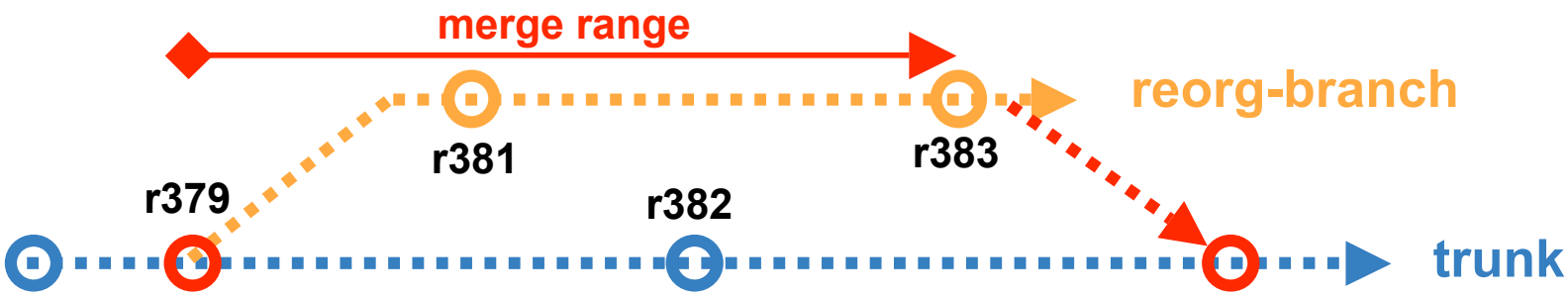
```
cd <branch-workcopy>
svn merge -r 381:382 <url>/proj1/trunk
svn commit -m "…"
```

undoing a already committed change (e.g. r328 was a bad commit)

```
svn merge -r 382:381
svn commit -m "…"
```



tutorial: merging an experimental branch



1. create a branch …

```
svn copy file:///c:/svn_rep/jaws/trunk \
file:///c:/svn_rep/jaws/branches/reorg-branch \
        -m "created reorg-branch from trunk"
```

2. check out new branch …

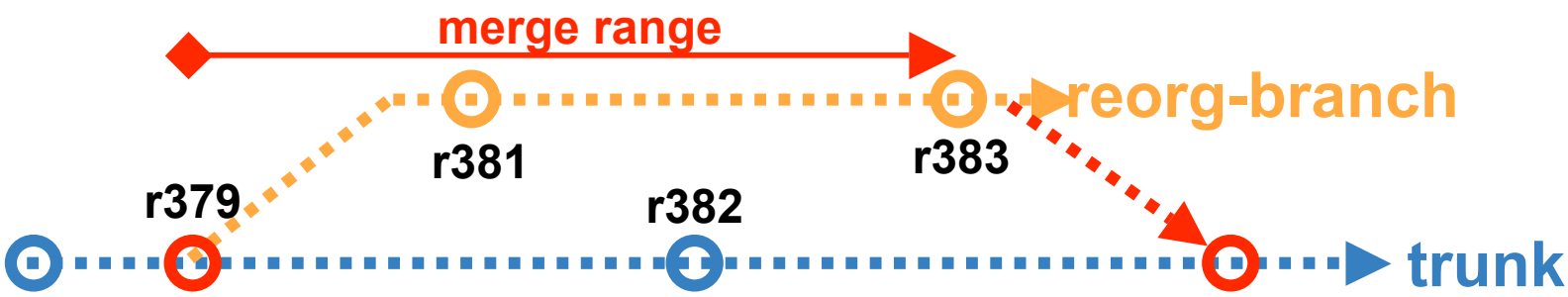
```
svn checkout file:///c:/svn_rep/jaws/branches/reorg-branch \
        jaws-reorg
```

3. do changes …

4. (if not tagged,) find revision when branch was created (last entry):

```
svn log --verbose --stop-on-copy
file:///c:/svn_rep/jaws/branches/reorg-branch
```

tutorial: merging an experimental branch (2)



4. merging…

`cd jaws` (change to trunk working copy)

`svn update` (ensure that the working copy is up-to-date)

```
> At revision 383
```

```
svn merge -r 379:HEAD \
      file:///c:/svn_rep/jaws/branches/reorg-branch
```

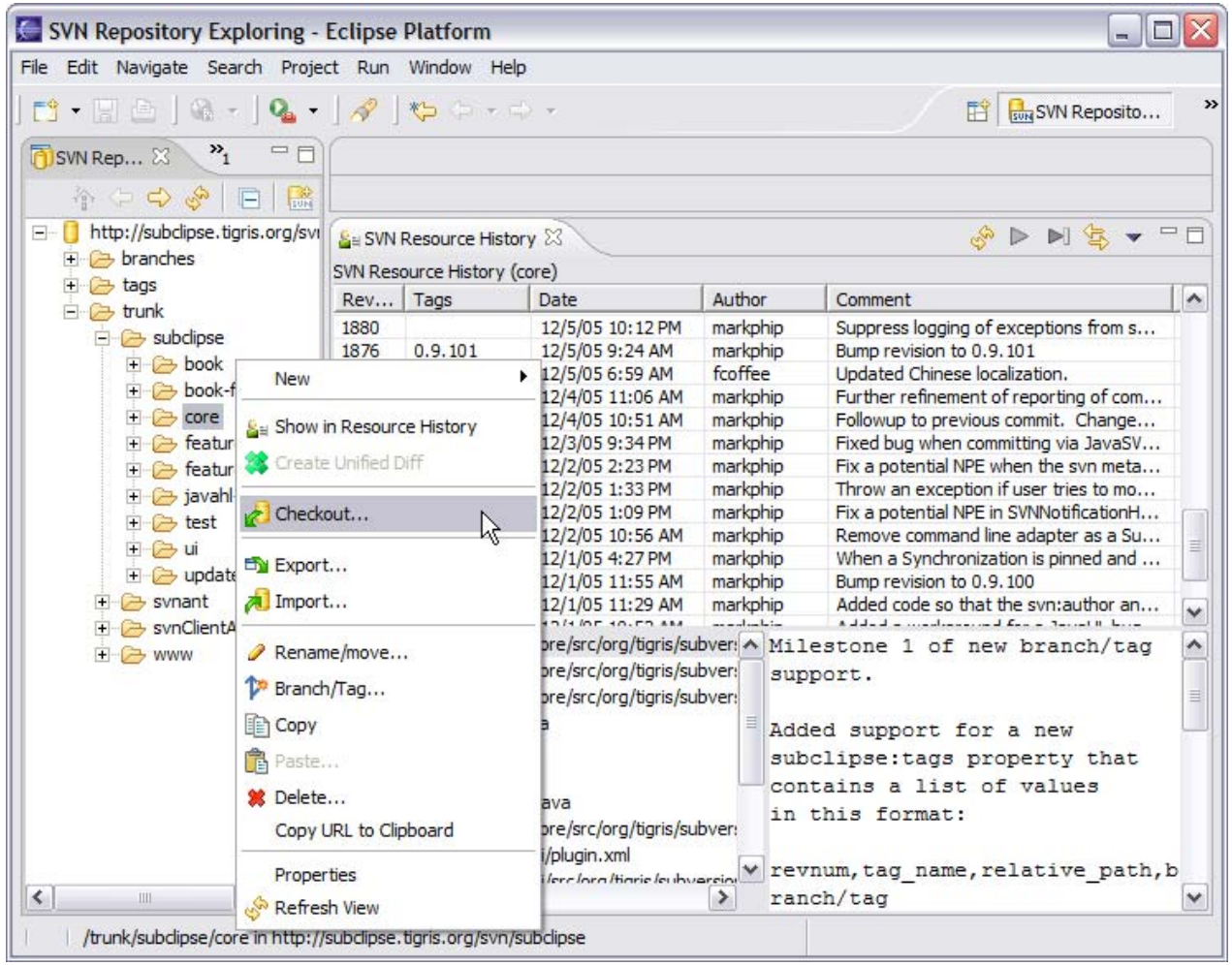
5. examine, compile, test, …

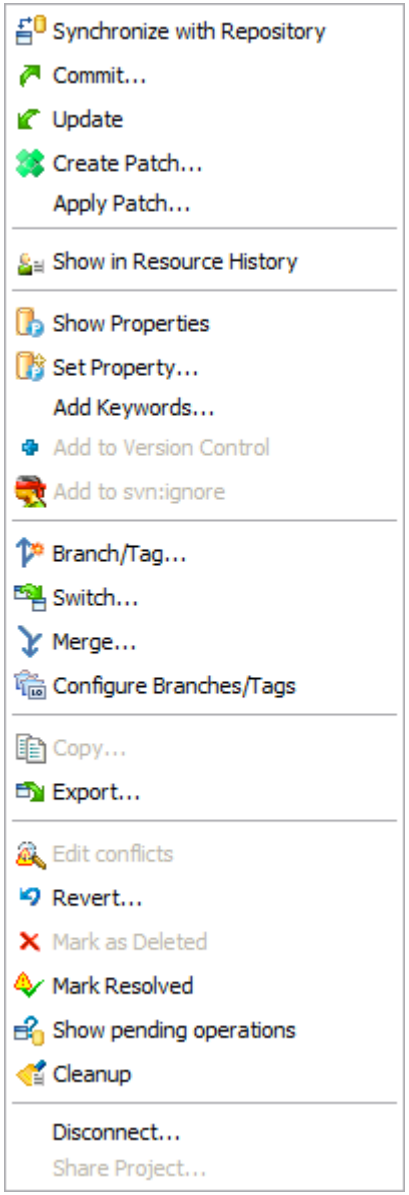
6. commit…

Always provide this information, when merging !

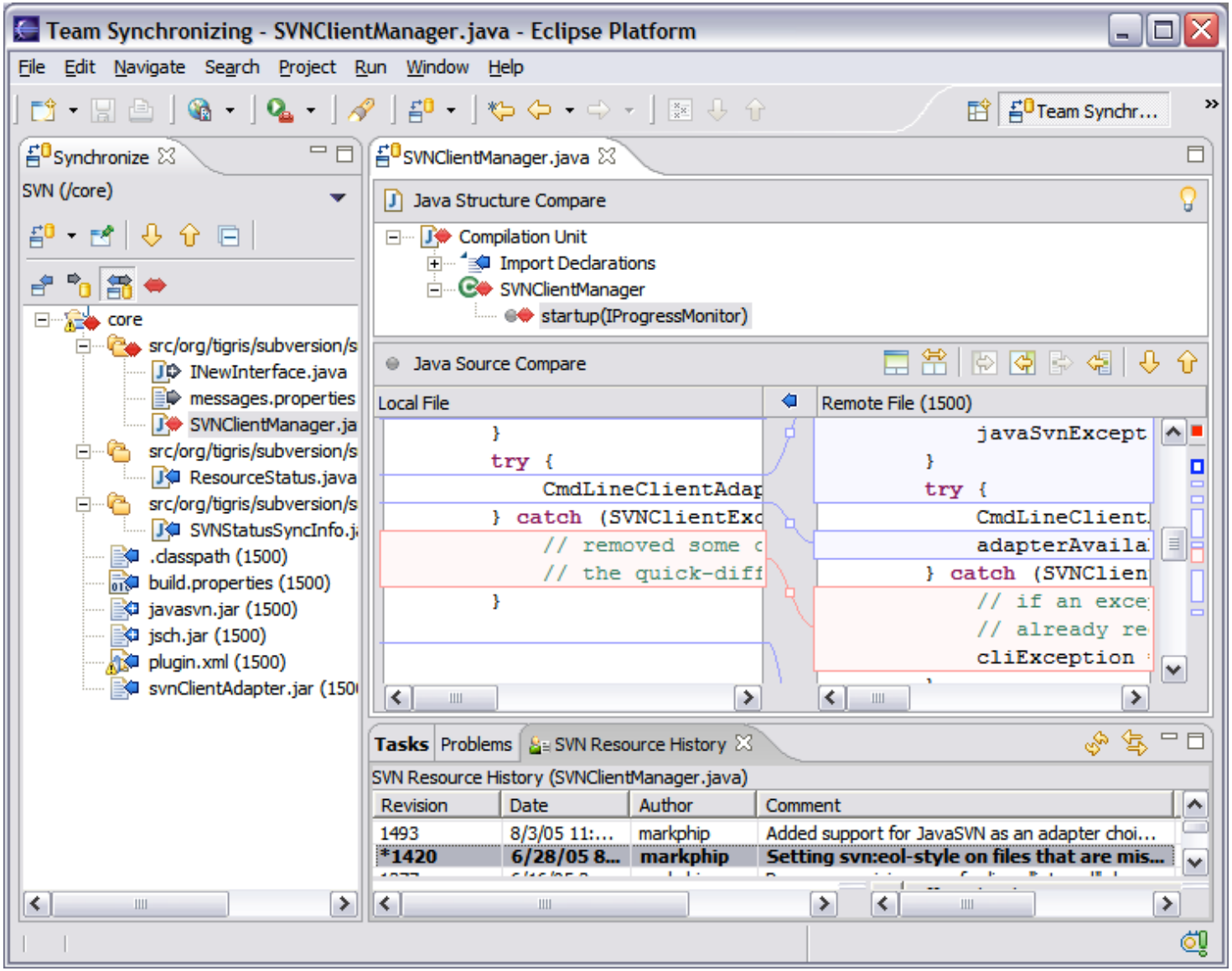
```
svn commit \
      -m "merged reorg-branch changes r379:383 into trunk"
```

IDE integration: subclipse





IDE integration: subclipse



recommended reading

- books
  - “The pragmatic programmer”, Andy Hunt and Dave Thomas, Addison Wesley
  - “Version Control with Subversion”, Sussman, Fitzpatrick and Pilato, freely available at http://svnbook.red-bean.com
  - “Pragmatic project automation”, Mike Clark, Pragmatic Bookshelf
  - “Pragmatic Version Control Using Subversion”, Pragmatic Bookshelf
- websites
  - www.pragmaticprogrammer.com
  - www.objectmentor.com